ESD-TR-91-222

MTR-11074

A Design for a Multi-Use Object Editor with Connections

By

D. S. Rand

October 1991

Prepared for

Director, SIGINT and Training Systems
Electronic Systems Division
Air Force Systems Command
United States Air Force
Hanscom Air Force Base, Massachusetts

Project No. 6290

Prepared by

The MITRE Corporation
Bedford, Massachusetts

Contract No. F19628-89-C-0001

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

[illegible]

EUGENE B. MORRIS, JR., CAPTAIN, USAF
Deputy Program Manager,
SENTINEL BRIGHT II

[illegible]

DAVID B. HULSLANDER, MAJOR, USAF
Program Manager,
SENTINEL BRIGHT II

FOR THE COMMANDER

Ronald A. Mason

RONALD A. MASON, GM-15
Director, SIGINT and Training Systems

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE October 1991	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE A Design for a Multi-Use Object Editor with Connections	5. FUNDING NUMBERS F19628-89-C-0001 6290
6. AUTHOR(S) Rand, Douglas S.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation Burlington Road Bedford, MA 01730	8. PERFORMING ORGANIZATION REPORT NUMBER MTR-11074
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Director, SIGINT and Training Systems (ESD/ICS) Electronic Systems Division, AFSC Hanscom AFB, MA 01731-5000	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESD-TR-91-222
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

One product of the Intelligent Courseware Authoring Tools project is a storyboarding tool. The foundation of this tool is a more general object editor with connectors. The methodology used is appropriate for a variety of object manipulations by direct user input.

The tool is intended to aid in the improvement of the courseware development process and facilitate the documentation and maintenance of completed courseware.

This document is a design specification which describes the architecture, design, and implementation of the storyboarding tool. A user's manual will be forthcoming under separate cover.

14. SUBJECT TERMS Authoring Tools Flowcharting	Storyboarding X Windows		15. NUMBER OF PAGES 41 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

ACKNOWLEDGMENTS

This document has been prepared by The MITRE Corporation under Project No. 6290, Contract No. F19628-89-C-0001. The contract is sponsored by the Electronic Systems Division, Air Force Systems Command, United States Air Force, Hanscom Air Force Base, Massachusetts 01731-5000.

In addition, the work performed for the Intelligent Courseware Authoring Tools project was funded by the Office of the Secretary of Defense.

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1
INTRODUCTION

This document is a design specification for the storyboard tool produced as part of the Intelligent Courseware Authoring Tools project. It is intended for technical staff who are familiar with the basics of programming and using a window system such as the X Window System. It documents the architecture, design, and implementation of the particular tool with some emphasis on how some of the techniques can be reused for other applications.

The general problem was to help courseware authors do a better job with less manual labor. They currently do not have appropriate tools available on their development workstations to flowchart their courseware. Some commercial tools exist, but are part of larger, expensive computer-aided software engineering (CASE) tool suites. Additionally, a custom tool can later become a knowledge-based tool to provide a front end to database navigation and courseware design.

Designing and building a specialized flowcharting tool alone would not improve our user's set of tools. They already have a flowcharter which runs on the IBM PC, and since we wish to improve upon their current tools, we must go beyond the basic flowcharting tool. The requirement for future extension made an open-ended design important.

The resulting tool is user extensible and compatible with our future plans for next year. It provides both a flowchart type interface along with more storyboarding information held inside iconic containers.

The fundamental nature of a direct manipulation interface is well suited to object-oriented programming and so this is the paradigm chosen for the architecture and implementation.

The sections are divided according to the conceptual level of the description. Logically, we first describe the architecture, the classes, the direct manipulation interface, implementation details, and finish with some walkthrough. At the end of the report is a summary on the current implementation state.

The architecture described in the next section is key to understanding the overall strategy of the application.

SECTION 2

ARCHITECTURE

2.1 INTRODUCTION

The storyboard tool uses the object-oriented programming paradigm. In this paradigm, one distributes the definition of functions amongst the various classes, and uses the class of the object being manipulated to determine which of many function methods is the correct one to evaluate. While this paper is not intended as a tutorial on object-oriented programming, it will introduce some basic terms and concepts in terms of conventional programming.

The choice of C++ as an implementation language gives the best combination of portability while retaining features of an object-oriented language.

class A structure description. Unlike conventional structures, the resulting class instances (see below) are composed of the sum of their class and of their parent classes. The classes form a tree. So each class has a set of instance and class variables of its own and all of its parent's instance and class variables also.

class variables In C++ these are called static members. Class variables have only one copy per class. That is, every instance has a variable whose value is the same, and this variable can be modified through the access of any instance. They are used for tasks such as recording how many objects have been allocated, and other class level state information. In conventional programming these would be external variables or data commons.

generic function The entire set of methods which comprise the implementation of a specific function over all classes. Some object systems call this a *message*.

inheritance The concept that a particular class' instance variables, class variables, and methods are the combination of all of the parent classes plus the changes that the class introduces. This is responsible for both the storage of each class being the combination of itself and its parent classes and also the notion that the method most specific to a particular class implements the particular generic function.

instance One instantiated copy of a specific class, an allocated block of memory, the correct size for the specific class, and initialized according to the class description.

instance variables In C++ these are called members. Instance variables are simply the structure components that belong to each instance.

method In C++ these are called member functions. A method is the particular part of the generic function pertinent to a specific class. So the draw method for a circle object would draw a circle and the draw method for a square object would draw a square. The combination of the circle draw method and the square draw method is the generic function draw.

method combination This is the incremental programming aspect of the object-oriented programming paradigm. A method can access its parent's method to perform all but the specific class-dependent programming. So a class circle-with-line-in-it could use the circle class' draw method to do everything but the extra line.

The methods for a particular class form an interface to the class. Most object programming languages also feature some data hiding to make the interface more effective. The overall effect is that good modularization is somewhat easier to achieve than in conventional languages like C or PL/1.

The following is a C++ convention which one needs to know about to read later sections. When we make a new instance of a class we say that we run the ***constructor*** for the class. When we destroy a class instance we run the ***destructor*** for the class. These methods allow the programmer to do bookkeeping and initialization on the class instances.

The actual behavior of an application design with the object-oriented paradigm is strongly tied to the class hierarchy of the class descriptions and the functionality contained in each class.

2.2 CLASS HIERARCHY

The class ***graphical*** is an abstract[1] base class for the direct manipulation interface. Each instantiated class is related to the class ***graphical*** through inheritance. The graphical class includes the generic information about each object such as its label, description, and connectivity with other objects.

The tool manipulates objects derived from class ***graphical***. The class ***graphical*** supplies a well-defined interface which all of its subclasses use. The class hierarchy describes the class/subclass relationship between all the classes. The class hierarchy of ***graphical*** is presented in figure 1.

[1] Abstract classes are never instantiated, they form a basis for other instantiated classes.

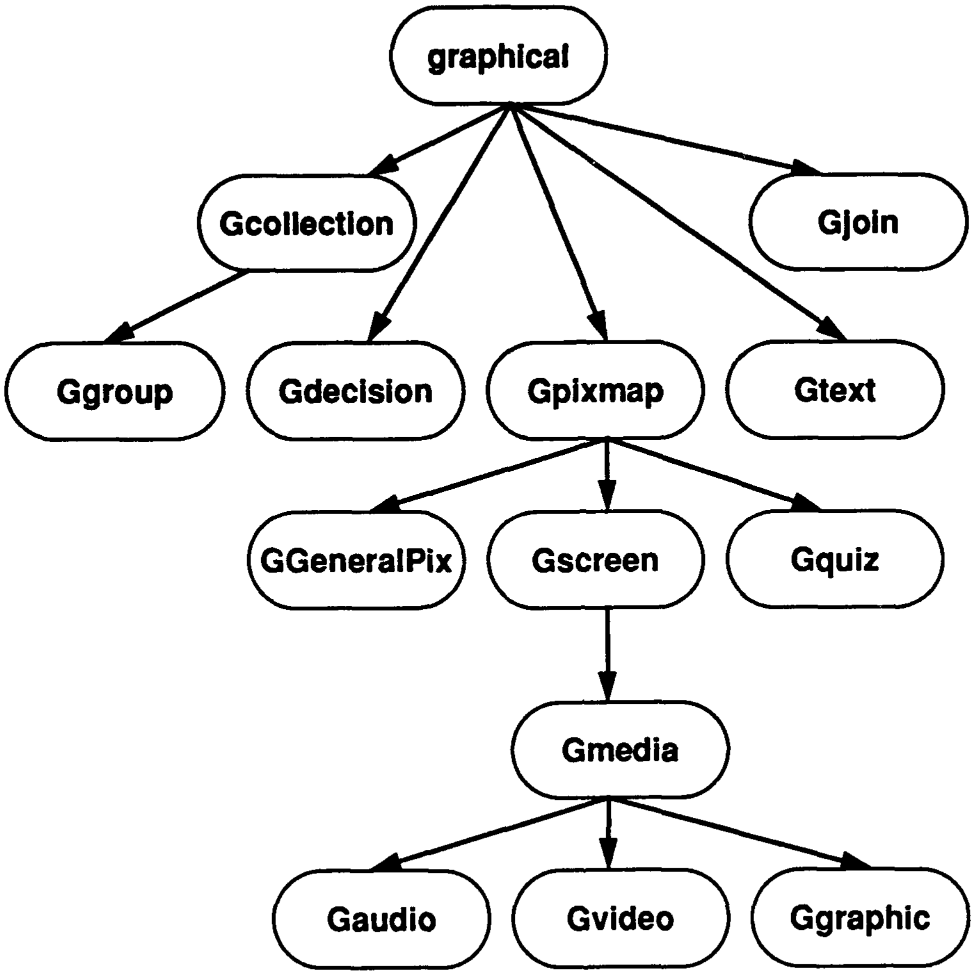

Figure 1. Class Hierarchy

Other utility classes include class ***List*** which is a generic list of graphicals and class ***connection*** which describes connectivity between objects. Connections are indirectly manipulated by the user.

2.3 USER INTERFACE

The user interface for the application can be divided into two sections: a Motif-based interface that provides for tasks like filing and editing, and a direct manipulation canvas that allows the user to move and edit the graphical objects directly.

The user interface technology for the direct manipulation interface is Xlib. The application has a single canvas upon which all the graphical objects and their connections are drawn. The user manipulates the objects by selecting, moving, and editing the objects on the canvas. All these manipulations are implemented by the application and not by the use of the toolkit. The general application interface layout is shown in figure 2.

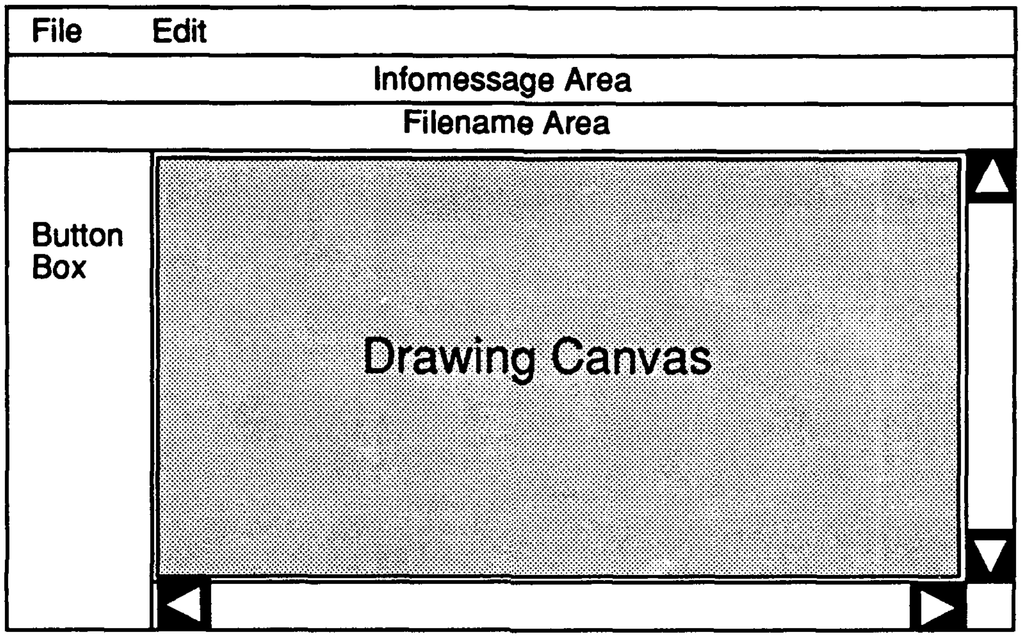

Figure 2. User Interface

The Motif widget set is used for the application user interface. A full description of all the widget classes[2] can be found in the "OSF/Motif™ Programmer's Guide." There are a number of user interface features using Motif. On the top of the application window is a MenuBar with **FILE** and **EDIT** menus. The file and edit menus each get the obvious functionality.

The second two areas are a Label widget which serves as an information message area right below the MenuBar and directly below the Label widget is another Label widget for the current file name and the current layer of the flowchart.

Remaining are two areas: to the left is a RowColumn widget for buttons arranged in columns, and to the right is the scrolled DrawingArea. The DrawingArea widget is not scrolled by the ScrolledWindow widget but rather the routines that display on the fixed DrawingArea widget are controlled by the ScrollBar widgets.

[2] Not to be confused with the application's class hierarchy.

The RowColumn widget is used for buttons for more commonly used functions than those tied to buttons in the MenuBar. Here are buttons for resetting the state, pushing and popping levels of Gcollection[3] objects, and selecting what object to place next.

The mouse buttons are used to interact with the canvas. This part of the interface has only minimal interaction with Motif or the X Toolkit. The available button actions are:

Single click left
If new object selected from buttons, then place new object; otherwise, select object pointer is on.

Double click left
Edit object pointer is on.

Shift click left or click right
Paste current selection from X Windows. This allows the user to paste in text as a text object from a text editor and also allows for a simpler duplication interface, allowing the user to click left on an object and shift click left to place a duplicate.

Drag left
If no object pointed at, then rubber band a multiple selection box. When the user releases the button, the resulting rubber band box is used to select a group of objects. If over an object currently, then a connection is attempted.

Drag middle
Move object or current group pointed at. The highlight outline of the object or group is dragged and the object moved when the mouse is released.

2.4 PRINTING

Printing is accomplished by having printing routines equivalent to the X drawing routines. This is basically a WYSIWYG[4] editor with some exceptions. There are some slight differences between the screen and the printed representation. For example, the screen shows an arrow with two lines and the printer with a filled arrowhead.

[3] Gcollection objects are explained later in the text. They are graphicals that contain a list of graphicals to provide encapsulation of detail inside a single object.

[4] What you see is what you get.

One nice correspondence is the way connections are shown on-screen. Bezier curves are used for this and are exactly the same on the screen as on the printer. In addition this is a fine solution for the difficult problem of making the diagram look neat and uncluttered.

2.5 CONNECTION DRAWING

Other solutions were tried including straight lines, simple lines, and more complex Bezier curves, but the best solution was to draw a simple Bezier curve whose size is proportional to the distance between the two points to be connected. Overall, the nicest looking curves resulted when the proportion was 1/2. In general, only one Bezier curve is used except when the start and end points are in line with one another. Here a single Bezier curve would result in a straight or nearly straight line. So we draw two curves. The first is a dog leg around the start object and the second is identical to all the other curves we draw.

2.6 FILE I/O

Each object is written to file in ASCII. There was a decision to make the files transportable, which meant avoiding binary files. Other advantages of using simple ASCII is the ability to extend the file format later and not invalidate old files.

When writing out to file, the objects are renumbered (they are internally numbered for identification) by walking the tree of objects. Then they are individually written to file. The connection information is written with them. Each connection describes the identifier[5] of the connected object and the connector in use.[6]

While not essential to renumber, it is convenient when reading the objects back in. The same function can read in a fresh chart and insert data into an existing chart. This is because the identifiers are all that is used in determining connectivity. So we can use the last identifier allocated to assure that there is no conflict with read-in objects; that is to say, we can add a constant offset to the inserted data object identifiers and not require complex schemes to fix up the connectivity information.

The classes form the basis of the modular programming in an object-oriented system. Practically all of the internal behavior of this application is determined by the class description and hierarchy.

[5] The instance variable ***id*** of class graphical.

[6] This is why we must renumber.

SECTION 3

CLASSES

3.1 CLASS GRAPHICAL

There are two program variables referred to in the following text. The variable `object` contains the current level being worked upon in the editor. The variable `topobject` contains the root of the tree of all objects being worked upon. The variable `topobject` is the only object of class ***graphical*** whose parent instance variable contains 0

3.1.1 Instance Variables of Class *graphical*

parent Each graphical object has a parent. There is a top level container object (of class ***Gcollection***) which is the top of the tree of graphical objects. The top level container is the only object in the tree with no parent (that is parent == 0).

id This is a unique id per graphical object. It is used in conjunction with saving and restoring graphicals in files and cutting, pasting, and copying graphicals.

x,y The position of the graphical on the canvas. This is in world coordinates.

height,width The apparent height and width of the graphical only. This does not include the length of the label string or connection paths. It is just the bounding box. For most graphicals these are fixed at creation time. This is, like the (x,y) position, in world coordinates.

label A character string labeling this graphical. It is initially set to an empty string.

description A character string (possibly very large) that contains the description for this graphical.

name The class name is stored in this string for file input/output purposes and also for debugging. It helps that all objects have such a label when examining objects in the debugger.

number This is a class variable that tracks the total number of instances of graphical. The only method that references this is the constructor for graphical. It is used for assigning ids.

nprotoconns The number of available connectors on this graphical. Currently this is 4 for all graphicals, which provides a connector on the top, bottom, and both sides of each graphical. Larger numbers might be useful for another type of application using this kind of interface to provide, for example, an integrated circuit with a dozen inputs and a dozen outputs.

protoconns An array of connector descriptions. See the class protoconn.

selected A flag indicating that the object is selected. It should only be on for one object at a time. There are a set of functions and member functions, which are always used to set this to avoid problems. Small pieces of code may avoid this for performance reasons.

conns Connections to other objects. See the class ***connection*** for a full description.

3.1.2 Methods of Class *graphical*

void virtual Draw() Draw the graphical on the canvas.

void virtual Print(file) Print the graphical in postscript on the given output file.

void virtual highlight(dohl) If dohl is 1 then highlight the graphical on the canvas, if dohl is 0 then unhighlight the graphical.

void virtual edit(x, y) Edit the graphical. Put its edit window at (x,y) on the display.

void virtual editbefore() This sets up any additional information this subclass of graphical needs to have edited.

void virtual editafter() This recovers changes for the additional fields that `editbefore` set up.

void virtual Show() Do something to display yourself. This is ignored by all non-media objects. In the current version only the class ***Ggraphic*** will do anything with this.

void virtual moveto(x, y) Move the graphical to another location.

void selectObject() Select this object.

int virtual select(x, y, x2 = 0, y2 = 0) If x2 and y2 are not given (which C++ allows with the =0. above defaulting their values to 0), then does a rectangle containing this object contain the point (x,y)? If x2 and y2 are given, then is this object's midpoint contained in the rectangle bounded by the points (x,y) and (x2,y2)?

void reparent(newparent) Reparents a graphical. All graphicals, except the variable `top-object`, are child objects.

int findconn(x, y) Find the connector on the graphical nearest (x,y).

void calcProtoconns() Create the prototype connectors for this graphical (part of initialization).

void fixconnections() Fix the connections after pasting a graphical.

void virtual readOn(int key, String value) Process instance-specific information during load.

void virtual storeOn(ostream) Store instance-specific information on the output file during save.

graphical virtual *findbyindex(id) Find the graphical with a matching id. This does a recursive descent from a Gcollection to find a contained graphical. It returns 0 if no matching graphical is found.

int virtual isCollection() Predicate that returns 1 if and only if the graphical is of class ***Gcollection***.

int virtual isGroup() Predicate that returns 1 if and only if the graphical is of class ***Ggroup***.

int virtual isMedia() Predicate that returns 1 if and only if the graphical is a subclass of class ***Gmedia***.

void virtual breakconns() Remove all connections on this graphical.

3.2 CLASS PICTURE

Class ***Picture*** is a simple static class used to store an X Pixmap and a PostScript bitmap of each read in bitmap. The constructors for the class do all the interesting work. The constructor allocates an X Pixmap (unless one already exists) and loads in the data from an external or internal source.

PostScript uses an ASCII string to represent a bitmap as an encoding of hexadecimal digits (i.e., each character in the string has an ASCII value of 0 ... 9 or A ... F, and represents a bit pattern from 0000 ... 1111 binary). So a character string is allocated which is filled with ASCII characters of hexadecimal digits to be used later when printing graphicals of class ***GPixmap*** or those derived from ***GPixmap*** on a PostScript printer.

3.2.1 Instance Variables of Class *Picture*

Generally, all the instance variables of the other classes described here are public—they may be accessed by anyone with a valid instance of the class. All of the instance variables of Class *Picture* are private except for `name`.

picts is an array class variable that stores all the instances so that they may be found using the function `lookupPic`.

number is a class variable that tracks how many Picture instances have been created as the array class variable `picts` has a fixed upper limit.

width, height stores the height and width of the image.

xpicture stores the X Pixmap associated with this instance.

ppicture stores the character string for PostScript associated with this instance.

name the name of this Picture. This is used for matches in the function `lookupPic`.

3.2.2 Friends of Class *Picture*

Friends are functions that have access to the private data of a Class.

drawimage draws the given Picture on the canvas.

printimage outputs the given Picture for the printer.

userbuttons initializes the user defined buttons and requires access to the Pixmap.

lookupPic(String) finds a Picture whose name matches the given string.

3.3 CLASS CONNECTION

Class ***connection*** helps manage the connectivity of graphicals. By providing a place to record drawing path information it provides a convenience by not requiring the output routines to recalculate this information continually. It also provides a nice abstraction from direct pointers.

3.3.1 Instance Variables of Class *connection*

from, to pointers to the graphical objects this connection connects.

fromindex, toindex these are the indices of the connections in the conns array of the from and to objects.

path this array contains the points for the Bezier curve to draw.

ncurves the number of curves to draw this connection.

arrowdir the direction of the ending arrowhead.

3.3.2 Methods of Class *connection*

~connection() removes the connection from the graphicals using it.

void calcpath() recalculates the drawn screen path.

3.4 CLASS PROTOCONN

Class ***protoconn*** is just a data structure to record information about the connectors (versus the connections) on a graphical. It has no methods.

3.4.1 Instance Variables of Class *protoconn*

x, y is the relative (x,y) position of the connector from the origin (x,y) of the graphical object.

cdir is the direction of the connection to this connector as one of the enumeration: DUp, DDown, DLeft, or DRight.

3.5 CLASS LIST

Class ***List*** is a simple, polymorphic, singly linked list of graphicals. Items are added as to a LISP type list by adding them to the front of the list. Items are removed by destructively modifying the list. This class is used both as lists in the class ***Gcollection***, the class ***Ggroup***, and as a stack for various processing tasks.

3.5.1 Instance Variables of Class *List*

object a pointer to the object contained in this instance variable.

next a pointer to the rest of the list, this is 0 for the last element of a list.

3.5.2 Methods of Class *List*

List(void *object, List *rest) creates a new List object with the specified object and whose tail is the *List* rest.

List(void *object) creates a new List with an empty tail.

~List() systematically deletes all the List nodes in this list. When removing a single node we carefully set that node's next field to zero which prevents this action.

List *remove(void* item) returns a List without the node whose object is equal to item (a pointer comparison).

int member(graphical* item) returns 1 if the item matches one of the objects on the List and 0 otherwise.

3.6 CLASS SELECT

Class *Select* is the state holder for the application and manages new objects.

3.6.1 Instance Variables of Class *Select*

int state The current state of the application.

graphical *data A holding place for new graphical objects prior to being put in place on the canvas.

3.6.2 Methods of Class *Select*

void dispose() makes sure that the contained data is deleted if and only if it is not null.

3.6.3 Application States

SSelect for selecting an object.

SNew for having a new object created and ready for placement in the chart.

SMove currently moving an object.

SConnect trying to connect two objects.

SMselect doing a multiple selection.

3.7 CLASS CLIPBOARD

Class ***Clipboard*** is the clipboard for the application. It keeps track of cut status, whether the object has ever been pasted, and such. Note that this is not directly related to cutting and pasting between applications.

3.8 CLASS CRESTORE

Class ***crestore*** holds a connection until all the objects have been read in from the file. It is then used to restore the connections. A connection has had its direct pointers to objects reduced to the object ids. These are restored by finding the objects matching the ids.

3.9 CONCLUSIONS

The classes which make up the application define and constrain the design in strong ways. This has always been true of the choices of data structures in programs, but in object-oriented programming this becomes even more pointed. But, while the data structures or classes define and constrain the internals of the design, the look and feel, the man-machine interface of the application, are a separate issue.

The look and feel of the chart and the way the user interacts with the application is determined by both the widget set, for which we use OSF/Motif^{tm}, and by the direct manipulation interface. You might even consider the user interface to be an entirely separate design. This is the subject of the next section.

SECTION 4

INTERACTIVE GRAPHICS

4.1 INTRODUCTION

The classes describe on the internal level what the data structures are and how they hold together, but the application is more than this. The application presents a view of the internal data structure as graphics which the user can directly manipulate. These graphics are the concern of the `Draw` and `Print` methods on objects of class ***graphical***. However, there is a whole graphics subsystem in charge of displaying and printing the objects which goes beyond the basic methods.

An important design decision revolved around the combination of interactive graphics and drawing libraries. Traditional libraries, like GKS, have limited user interface models, and the mechanisms with good user interface models are primarily meant for other kinds of interfaces and don't include scalable or translatable graphics. So it was necessary to create a simple graphics system to deal with issues of world versus device coordinates.

4.2 WORLD AND DEVICE COORDINATES

As in traditional graphics systems, the graphicals are maintained in a world coordinate space. All events are translated from device coordinates to world coordinates, and all graphical outputs are translated from world coordinates to device coordinates. In figure 3, the translation of world coordinates to device coordinates is shown. Note that the while people refer to the "X Window System" the proper name for the output area on the screen is a ***viewport*** and the area being shown is the ***window***.

In the current system, we only provide scrolling of output on the canvas. We could also provide scaling of the graphics output but do not because of the issues of font and Pixmap scaling. In particular, there is no support mechanism for this in X—even to scale by powers of 2.

4.3 GRAPHICS PRIMITIVES

For drawing, we define a set of primitives which abstract the interface from the `Draw` methods of each object of class ***graphical***. In this way, we simplify the methods and allow ourselves to replace more easily the current mechanism for drawing in the future. We also take the issues of device versus world coordinates out of the specific methods.

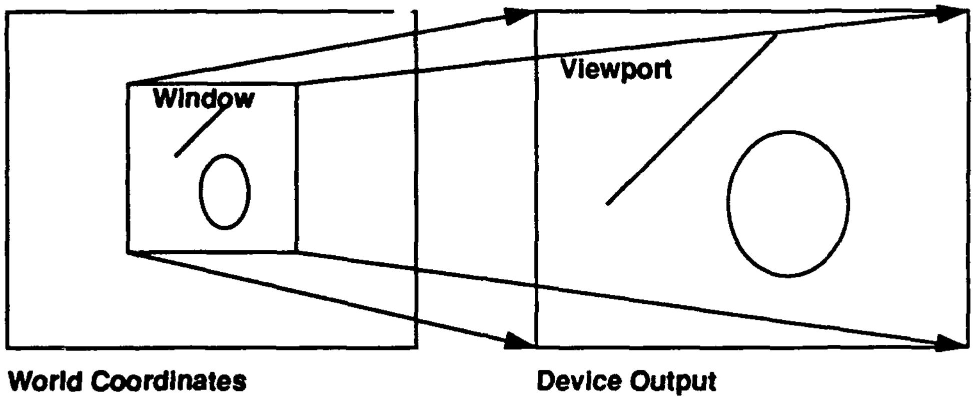

Figure 3. Translation and Scaling of Graphics

The basic drawing primitives are:

DCtoWC(int x, int y, int *nx, int *ny) Translates a device coordinate (x,y) into a world coordinate (nx,ny).

WCtoDC(int x, int y, int *nx, int *ny) Translates a world coordinate (x,y) into a device coordinate (nx,ny).

drawbox(int x, int y, int width, int height) Draws a box at (x,y) with (width,height).

drawline(int x1, int y1, int x2, int y2) Draws a single line from (x1,y1) to (x2,y2).

drawlines(int *x, int *y, int npoints) Draws the polyline contained in the x and y arrays.

drawcircle(int x, int y, int diameter) Draws a circle whose bounding rectangle has its upper left corner at (x,y) and the given diameter.

drawtext(int x, int y, String text) Draws the given text at (x,y). The text is positioned so that its bounding box has its upper left corner at (x,y).

drawimage(int x, int y, Picture *pict) Draws the given pict's Pixmap onto the screen at (x,y).

In printing, we handle the basic primitives in a different way. We define a series of Postscript routines which handle the minor extensions to what Postscript has predefined. These extensions are:

box x y width height Draws a box.

circle x y diameter Draws a circle.

arrow r s x y Draws an arrowhead of size r at angle s at (x,y).

While we cannot conveniently scale the screen output, there is a printing mechanism in PostScript that will take care of scaling and translation. So, in printing, we just check the bounds of the diagram to be printed and scale the diagram accordingly.

The only interesting issue in providing PostScript output is handling the relative coordinate systems of X and PostScript. In X the point (0,0) is at the top left of the screen. In PostScript the point (0,0) is at the bottom left of the page. To translate between, we give PostScript a translation to move the page's origin to the top left and then invert all the y coordinates on output. So a graphic which appears at (10,10) on the screen is placed at (10,-10) on the page. This is an adequate solution.

The actual implementation of the direct manipulation interface is in the guts of handling events from the X server. This is the subject of the next section.

SECTION 5
IMPLEMENTATION DETAILS

5.1 EVENT HANDLING

The heart of the direct manipulation interface lies in event handling. For the other part of the user interface, the Motif widgets, event handling takes care of itself and we only need to add appropriate callbacks for buttons and scrollbars. But the direct manipulation of the objects of class ***graphical*** is more like the internals of a widget.

Event handling is a tricky business. There are many events one can choose to handle. For this application's canvas, we limit ourselves to button events and exposure events.

When the user presses any mouse button there is a "passive grab" of the server. This means that if a user presses a button we are guaranteed to get both the button press and the button release events. This is crucial to how the direct manipulation interface works.

We have three callbacks for button events on the canvas, one for each of button press, button move, and button release events.

5.1.1 Button Press Event

We first discard any event for the right button. The middle button is always used for moves and the left button is used for several things. If the middle button is pressed and there is no currently selected item, or there is a currently selected item and we aren't pointing at it, then we select a new item before starting to move it. We only change to the move state if we succeed at selecting a new object.

If the left button is pressed and we have a pending object to be placed, then we place the object in the indicated location, otherwise we do a selection. If the left button was pressed very recently (within the last 350 milliseconds), then we consider it to be a double click and we run the object's edit method.

When the object is selected it is highlighted.

There is a state variable which shows the current state, and the canvas cursor mirrors the state to visually cue the user. The states and associated cursor cues are as follows.

State (Internal name)	Cursor
Selection (SSelect and SMselect)	arrow
Moving (SMove)	hand
Placing new (SNew)	crosshair
Connecting (SConnect)	dot

5.1.2 Button Move Event

During the last button press the state was set to either `SSelect`, `SMove`, or `SMselect` (multiple object selection).

If the state is `SMove`, then we start dragging the object's drag box around. We do this by unhighlighting the box, and then rehighlighting in the new cursor position. The result is that the object's box tracks the cursor. The box is a simple rectangle and is managed by the dragger routines in `dragger.cc`.

If the state is `SSelect`, then we check if the user is dragging the cursor. If the user is dragging the cursor, then he is attempting a connection to another object. We change the state to `SConnect` and put up an appropriate message.

If the state is `SMselect`, then we are drawing a rubber band box on the screen which will be used in the button up event handler to select a set of objects.

To get smooth animation, for the rubber band box, we do an `XSync()` after each change.

5.1.3 Button Release Event

This is where we finish off all the set up actions from the press and move events.

If we were moving the object, then we finish the move, reset the state back to select, and redraw the canvas. The final cleanup includes redrawing all the connections attached to the moved graphical.

If we were doing a multiple select, then we now call the selection routine with the final rectangle drawn.

If we were trying a connect, we look up the connectors and do the connection.

5.2 CANVAS MANAGEMENT

The current canvas is 11" x 14" and is of fixed size. The application doesn't yet do automatic scrollbar sizing or different size canvases. This is not seen as a crucial extension of functionality.

The redisplay algorithm has several paths. The canvas can be redisplayed for any canvas area in whole or part . Only visible objects are redrawn in this case. The algorithm can also do a full redisplay. Both of these options are available with or without a clear area.

In general, we try to avoid a redisplay call. On a paste, we just redraw the object and any connections drawn to or from. On a move, we erase the old object position by a redisplay with a clear area, and then draw the object in its new position. Full redisplays mostly happen from de-iconification of the application or some other exposure event.

Scrolling is currently handled by the graphics software, but another mechanism could take over the scrolling of the canvas. The scrolling window widget that contains the drawing area widget is capable of scrolling its child widget. This still doesn't deal with issues of scaling but it is a notion for dealing with a fair amount of translation that currently happens.

5.3 WIDGET CREATION AND MANAGEMENT

There are three initialization files. `init.cc` does almost all the widget creation and setting up of callbacks. The dialog boxes are created in `dialog.cc` and also in `text.cc`.

Resources for the widgets are specified in the resource files, `app-defaults/FCEdit`,[7] `.Xdefaults`, and in the source code. It would have been preferable to place all of the resources in the application defaults file, but the form widget information cannot go there as there are problems with supplying the type conversion required by Motif.[8]

The dialog widgets are managed by using `XtManageChild`. and `XtUnmanageChild`. The normal intrinsics routines `XtPopup` and `XtPopdown` are not used by Motif.

As with a number of details of an X-based application, the dialog management appears somewhat asynchronous. It would be more true to say that it is broken into two pieces. The first

[7] The app-defaults directory is in the `/usr/lib/X11` directory.

[8] For some reason Motif wants a string to window converter which cannot be supplied because the widgets have been created but not displayed at the time the conversion occurs, and a widget has no window until it is displayed (realized). Motif should be able to use the string to widget converter which is available from the Xmu library.

piece activates the dialog which sets up the dialog's activate callbacks for the OK and Cancel buttons. The activate callbacks then take care of dialog cleanup and value changes.

5.4 SAVE FILE DETAILS

The objects are saved in a canonical form in the save file. An EBNF (extended Backus-Naur form) description of the save file is as follows:

```
file                := collection-object
object              := standard-object | collection-object
collection-object   := standard-object object* end
standard-object     := classname cr data* end
data                := key space value null cr
null                := ascii-byte-0
space               := ascii-space
cr                  := ascii-newline
end                 := ":end"
key                 := ascii-string
classname           := ascii-string
```

So each object has a line containing the class name of the object. This is followed by some number of lines containing a string naming the value (a key) and the value saved (as a string). The value is terminated by a null character which allows us to have multiline descriptions and such stored in this way. The object is ended by a single line with the string ":end" on it. The exception is the collection objects which then have zero or more objects following them with a ":end" ending the object list.

All data is stored this way. A side benefit is that the save files will never become out of date. If some information is discarded, then the data in the save file is ignored. If some new information is added then the default value will be used.

The only thing to be careful about in this approach is that if the semantic meaning or format of a particular field changes, then a new key must be chosen for the save, and a compatible reader must convert the old saved data, but this is a small price to pay for upward compatibility.

Further elucidation of this and other subjects may be found in the following section which contains partial walkthroughs of the application for several common user requests.

SECTION 6

WALKTHROUGHS

6.1 INTRODUCTION

The best way to make clear some of the more confusing implementation details is to walk through the steps that the application takes in dealing with some of the common user requests.

6.2 EDIT DIALOG

Let's use the edit dialog for the standard graphical as an example.

1. The user double clicks on an object. The canvas buttonpress callback `dobuttondown` calls the selected objects edit method
2. The edit method copies the label and description instance variables into the dialog's text widgets, it calls the class `editbefore.` method, it configures a number of toggle widgets for connection breaking, and calls the function XtManageChild to manage the dialog shell.
3. The user manipulates the dialog fields. Notice that this is not a modal dialog[9] and will not force user input, i.e. the user can continue to edit the diagram and ignore the dialog. In fact, the user can double click on another object and the dialog will be modified.
4. The user clicks the OK or Cancel button. The function `finish0edit` is called.
5. The function `finish0edit` takes down the dialog box. If the OK button was pressed, then the changes recorded in the box are done. This includes modifying the label and description and removing any connections toggled off and the `editafter` method is called for additional modifications.

6.3 FILE LOAD

1. Put up the file selection dialog which will call the function load with the selected file information.
2. Open the file.

[9] Modal dialogs are those that take over until the user completes or cancels them. The Apple Macintosh makes use of modal dialogs for a number of things such as print setup.

3. If this is a load (not an insert) clear all the existing objects.[10]
4. Now read the top level object. This object will contain all the other diagram objects.
5. Each object consists of an initial line containing its classname, zero or more lines containing instance data, and a line with the text :end.
6. Each of the data lines has a keyword followed by a single space character followed by a null terminated character string containing data.
7. All normal instances will have at least an id, (x,y) position, label, and description data.
8. We create the specified object class instance.
9. Each data line is read in. The keyword is converted to a unique number, and the data is read into a String object. The `readOn` method for the class is invoked with the keynumber and String as arguments. The correct instance variable is found and set. This includes the connection information.
10. If this is a Collection object then we also look for some number of subobjects following the :end line of the instance data.
11. Continue as above for the next object.
12. Now scan the saved connection information and connect the read in objects.
13. Redisplay the screen and close the input file.

6.4 FILE SAVE

1. Put up the file selection dialog and call save with the appropriate filename.
2. Renumber the objects to have continuous ids starting at 0.
3. Now save the top level object which will save all its children.
4. For each object, start by writing the id and (x,y) position.
5. Then write the label and description and the description terminator.
6. Write out the connection information.
7. Then call the object's `storeOn` method to write out class-specific information.
8. Continue with the next object as above.
9. Close the file and tell the user the file is saved.

[10] This needs to be modified to allow the user to save the existing file first.

6.5 CUT

1. First a copy is made of the selection (see copy below).
2. Then the `clipboard.pasted` flag is set to false.
3. The connections are broken.
4. If this is a class ***Ggroup*** object, then we scan the list of subobjects in a nonrecursive fashion and break their connections.
5. Remove the item from the current level's list.
6. Set the modified flag.
7. Redraw the screen.

6.6 COPY

1. Remove the existing clipboard object if not null and unpasted.
2. Set up the clipboard structure.

6.7 PASTE

1. If the clipboard item is null, then return.
2. If the clipboard item has been pasted before, we do a deep copy of the object. This means we copy the entire structure of a Gcollection. We do not at this time try to duplicate the connections.
3. If the item to be pasted is a Ggroup, then we only paste its children one by one.
4. Otherwise we simply paste the object.
5. Then we fix the connections to all objects which were pasted.

6.8 PRINT

1. First we output a PostScript setup header which sets up the print font, scaling and translation for our output.
2. We define PostScript routines for boxes, circles, and arrows.
3. We do the following for each Gcollection including the top level. Gcollection:

   Output the translation setup

Output a header with the Gcollection's label.
Calculate the total size of this Gcollection.
Scale this pages output to fit the Gcollection.
Print the Gcollection's objects.
Start a new page.
Output the non-null descriptions from the Gcollection's objects.

SECTION 7
CURRENT IMPLEMENTATION

The current implementation is a growing, evolving application. Some current notes are here on future work planned and limitations of the current implementation.

7.1 FUTURE WORK

We hope in the next year to make this a more competent storyboarding environment. We will do this by adding integration with a media database and providing full support for viewing media objects contained in the storyboard tool.

7.2 LIMITATIONS

This is a partial list of current limitations of the implementation.

- Currently the user can scroll the canvas (DrawingArea widget) but not scale it. This is a current compromise as X doesn't provide scalable fonts or scalable Pixmaps. There is a relatively easy work around by allowing only power of 2 scaling, i.e., 2x, 1x, 1/2x, 1/4x, etc., but this would have been somewhat time consuming for the benefit.
- The canvas is of fixed size.
- Media objects other than Ggraphic do not yet have viewers.

BIBLIOGRAPHY

Open Software Foundation, 1990, *OSF/Motif™ Programmer's Reference*, Englewood Cliffs, NJ: Prentice-Hall.

Nye, A., 1988, *Xlib Programming Manual*, Sebastapol, CA: O'Reilly and Associates Inc.

Nye, A., 1988, *Xlib Reference Manual*, Sebastapol, CA: O'Reilly and Associates Inc.

Nye, A. and O'Reilly, T., 1990, *X Toolkit Programming Manual for X Version 11*, Sebastapol, CA: O'Reilly and Associates Inc.

Nye, A. and O'Reilly, T., 1990, *X Toolkit Reference Manual for X Version 11*, Sebastapol, CA: O'Reilly and Associates Inc.